13. ročník
Žilina 1. – 3. 7. 2025
Otvorený softvér vo vzdelávaní, výskume a v IT riešeniach
ossconf.fri.uniza.sk
OSSConf

OSSConf 2025: 1–8

# O jedné mapě na zakázku, část 5: Poznámky k optimalizaci kódu

PAVEL STŘÍŽ (CZ)

**Abstrakt.** Článek shrnuje několik tipů a triků u optimalizace zdrojového kódu. Drobné ukázky jsou v Pythonu, ve skutečnosti se však užívá C/C++. Zároveň představuje náklady na virtuální CPU a GPU na `https://compute.cudo.org`.

**Klíčová slova.** Distribuované výpočty, optimalizace kódu, předvýpočty, program bez větví, C++, R, Python.

## ABOUT PREPARATION OF ONE MAP, PART 5: NOTES ON CODE OPTIMIZATION

**Abstract.** The paper presents several tips and tricks on source code optimization. Snippets are presented in Python, even if C/C++ is actually used. It also presents prices of vCPU and vGPU at `https://compute.cudo.org`.

**Keywords.** Distributed computing, code optimization, precomputation, branchless programming, C++, R, Python.

## 1. Proč optimalizovat své kódy?

Odpověď na otázku je, že jde o čas, tedy o peníze. Ať už ušetřené či vydělané.

Na závěr pentaptychu o mapách bych rád upozornil na optimalizaci kódu. Především na oblasti, se kterými jsem se za poslední rok setkal, ale hlavně na to, že něco takového existuje, což u běžného denního programování člověka nenapadne.

Během sledování videa *Blazingly Fast,* `https://www.youtube.com/watch?v=hp4Nlf8IRgs`, jsem si říkal, že jsem pár triků už též použil, že by bylo záhodno se o zkušenosti podělit.

## 2. Měření času a výkonu

Možností, jak si přidat časovou známku do programu je mnoho. V Pythonu například knihovna `time`, v C++ knihovna `chronos`.

```
import time
zacatek=time.time()
#[...]
print(time.time()-zacatek)
```

Lze užít zápis do souboru, pokud nechceme časové známky ve výpisech. Touto cestou měříme délku běhu programu, ale můžeme měřit určité úseky ap. Lze užít přesměrování `1>soubor-bezny.log 2>soubor-error.log` na trvalé uložení. Můžeme užít linuxový program `time` na měření celé délky doby běhu programu.

Můžeme během optimalizace kódu zkoumat výkon určité části. Nejen čas je důležitý, ale i kolik RAM paměti potřebujeme. Je dobrý i odhad, kolik místa budeme potřebovat na pevném disku. Kolik času nám zhruba zabere výpočet všeho na jednom jádru. To se pak hodí při odhadu při rozhození na víc výpočetních strojů.

Zmíním programy `vgrind`, `gdb` či `perf`. Užití je nad rámec tohoto příspěvku, je to komplexní oblast. Během rešerše jsem narazil na knihu *Visualizing Performance*, `https://raw.githubusercontent.com/samlee2015jp/cs_books/main/Systems.Performance.Enterprise.and.the.Cloud.2nd.Edition.2020.12.pdf` podpořenou přednáškou na *Flame Graphs*, `https://www.youtube.com/watch?v=VMpTU15rIZY`.

U Picatu je super, že se člověk může na jednom řádku přepínat mezi SMT, SAT, CP a MIP. A někdy ani není potřeba měřit čas: člověk zkusí jeden modul, spočítá se úloha za pár vteřin, zkusí jiný modul a trvá to věčnost. Za pár okamžiků člověk ví, co bude pro jeho úlohu lepší.

Na tomto místě dodám, že profesor Knuth intenzívně pracuje na Volume 4 TAOCP. V roce 2024 vyšel Fascicle 7: Constraint Satisfaction. Na úvod zkoumané oblasti mohu doporučit přednášku Romana Bartáka, `https://slideslive.com/38891388`.

## 3. Algoritmy a datové struktury

Jádro programování však tvoří zkoumání algoritmů a užité datové struktury. Mně při práci pomáhá metoda půlení, ta je poměrně známá. Ve 2D jsou alternativou kvadrantové stromy (quadtrees), ve 3D pak oktálové stromy (octrees). Ty jsem užil při míchání rastrové a vektorové grafiky, viz samostatný článek. V Pythonu začátek vypadá zhruba takto:

```
uroven=1; ulozene={}
ulozene[uroven]={}
ulozene[uroven]["deti"]={}
```

Nebo přes zlepšovák v Pythonu (má vždy přednastavenou hodnotu):

```
from collections import defaultdict
uroven=1; ulozene=defaultdict()
ulozene[uroven]=defaultdict()
ulozene[uroven]["deti"]=defaultdict()
```

Plus se pracuje s rekurzivní funkcí, kdy přecházíme do hlubších úrovní.

U Rubikovy kostky, ale i při výpočtu počtu obarvení, viz dřívější části této série, jsem použil princip „meet in the middle“. Znamená to, že se snažíme řešit

problém z jedné i z opačné strany. Konkrétně k Rubikově kostce se hodil program GAP, `https://www.gap-system.org/`. Lze minimalizovat počet kroků k vyřešení Rubikovy kostky, pro hlubší studium doporučuji `https://www.youtube.com/watch?v=Q-6QToN62Rc` s podpůrnou stránkou `https://cw.fel.cvut.cz/b232/courses/ko/labs`.

## 4. Programovací jazyk

Je jich nespočet: C/C++, Python, R, Rust, Perl, Go, Java, JavaScript ad. Já se naučil na náročné výpočty užívat C/C++. Mohu doporučit přednášky *31 nooby C++ habits*, `https://www.youtube.com/watch?v=i_wDa2AS_8w` či *The Most Important Optimizations to Apply in Your C++ Programs*, `https://www.youtube.com/watch?v=qCjEN5XRzHc`.

Je dobré se naučit volat cizí programy jako knihovnu či modul. Spouštění externího programu je časově náročné. Jsem si toho vědom u generování zadání z řešení u obarvovací hry, kdy jsem použil Python, ten generoval soubor pro Picat. Picat pak musí takový soubor načíst, opět časová ztráta. Lepší by bylo zkoumané přímo naprogramovat v Picatu. V mém případě, kdyby to bylo v mých možnostech.

## 5. Práce s bity, průběžné ukládání, předvýpočty, vyhnutí se řazení, hashování

Zvláštní kategorii tvoří *práce s bity.* Je to efektivnější než práce s čísly. Je to jen trochu neintuitivní. Mohu doporučit knihu *Matters Computational: Ideas, Algorithms, Source Code* od Jörga Arndta, `https://www.jjj.de/fxt/fxtbook.pdf`. A to nejen na práci s bity, která je v úvodu knihy.

Na studium mohu doporučit užití bitových mask při generování sudoku, viz soubor `sudoku2.cc` na `http://www.afjarvis.org.uk/sudoku/bertram.html`. Chtělo to však papír a tužku a zorientovat se v tom.

Asi nejznámější příklad na *průběžné ukládání* je výpočet Fibonacciho čísel. Lze doporučit `https://www.youtube.com/watch?v=KzT9I1d-LlQ`. Vlastními slovy řečeno, výpočty, které se opakují, se za běhu programu snažíme uložit, znovu je nepočítat.

*Předvýpočty* se ukázaly jako zlatý důl co se úspory času týká. U úlohy maxima bloků v klasickém sudoku jsem například sčítal 9 čísel, devět 0 a 1. Abych se sčítání úplně vyhnul, nechal jsem si to předvypočítat, jedná se o $2^9$ možností. Je na to možné využít devítirozměrné pole.

Čeho jsem nevyužil: třeba u obarvovací hry sčítám dvě či tři barvy, kde barvy jsou reprezentovány čísly $1, 2, \ldots$. I to by se dalo do určitého počtu barev předvypočítat. Říkám tomu, že si připravuji tabulku malé či velké násobilky, jak to znají

děti ze základních škol. Počítače jsou šikovná kalkulačka, ale proč se výpočtům nevyhnout úplně, lze-li to.

Operace seřazení prvků je drahá. Někdy se jí dá vyhnout. Například chceme sečíst čísla 1, 2 a 5, ale nevíme, v jakém pořadí dorazí. Z kombinatoriky víme, že existuje 3! možností proházení. Opět lze takovou situaci předvypočítat.

Pro badatele zmíním datovou strukturu Hash Tables, ke které se snažíme dostat, je-li to možné. Na úvod by mohla posloužit tato přednáška, `https://www.youtube.com/watch?v=DMQ_HcNSOAI`.

Přikládám vzorek v Pythonu, fintu, kterou jsem se naučil od Štefana Pešky.

```
hodnoty={}
hodnoty[1,2,3]=66
vybrana=(2,3,1)
hodnoty[vybrana]=33
vybrana=[5,3,1]
hodnoty[*vybrana]=22
vybrana={7,5,6}
hodnoty[*vybrana]=11
```

Přes hvězdičku se nám hodnoty pěkně rozbalí a můžeme s tím pracovat. `(1, 2, 3): 66, (2, 3, 1): 33, (5, 3, 1): 22, (5, 6, 7): 11`.

Pokud jsem si nebyl v C++ jistý, kterou datovou strukturu mohu použít, tak jsem postupně zkoušel co se dalo. A dá se snadno měřit čas, pokud jich lze užít víc.

## 6. Práce s obřími čísly v C++

C++ je vysoce optimalizovaný programovací jazyk, ovšem práce s čísly nad `unsigned long long` alias `ull` je výzva.

Jeden z přístupů je z čísel přejít na text a operaci sčítání či násobení dělat jak na papíře, práce cifra po cifře. Je to časově náročné, ale možné.

Já jsem potřeboval v C++ jinou záležitost, o něco snazší situaci. Během rekurzivního běhu jsem si načítal počet řešení. Nebyl jsem si však jistý, kolik jich bude, teoreticky jsem mez `ull` mohl překročit.

Zde je školní příklad v Pythonu. Zkusme si, že nesmíme u součtu překročit hodnotu 1000. Načítáme dvojciferné hodnoty. Abychom u mezisoučtu nepřekročili 1000, nastavíme si mez o řád nižší (100). Maximum tedy je 99+99, což je tříciferné číslo. A zde je ukázka, kdy součet rozložíme na násobek čísel 100 (nad) a zbytek (pod).

```
mez=100 # O řád nižší než 1000.
nad=0; pod=0
for k in [99,89,67,72,80,92,49,76,55,98,96,84,47,88]:
  print(pod, "+", k, end=": ")
  pod=pod+k;
```

```
  nad=nad+pod//mez
  pod=pod%mez
  print(nad,"*",mez,"+",pod)
```

Na výstupu dostáváme:

```
0 + 99: 0 * 100 + 99
99 + 89: 1 * 100 + 88
[...]
57 + 47: 10 * 100 + 4
4 + 88: 10 * 100 + 92
```

Víme tedy, že součet je 1092, což by bylo přetečení 1000. Součet je rozložen na 10 (krát 100) a zbytek (92). Touto úvahou můžeme jít do pole, a kdybychom se báli, že proměnná `nad`, nyní 10, by mohla též přetéct, tak bychom si připravili proměnnou další (násobek 10000). Na závěr bychom v C++ použili buď práci s textovými řetězci, nebo předali výpočet Pythonu, kde se používá indexování cifer a práce s obřími čísly není problém, pokud to hardware zvládne.

## 7. Obejití větvení

Ať je nám to lidsky blízké a intuitivní, bohužel podmínky `if` a větvení `case` zpomalují náš program. Počítač je dobrý v počítání. To už říkával můj učitel informatiky na základní škole, pan Koutný. Používá se na to již odborný termín Branchless Programming, programování bez větví, kdy se snažíme podmínkám vyhnout. Mohu doporučit `https://www.youtube.com/watch?v=bVJ-mWWL7cE` či náročnější přednášku `https://www.youtube.com/watch?v=g-WPhYREFjk`.

Uvedu opět školní příklad. Pokud je jistá proměnná pod 5, k součtu přičtu $-1$, pokud je hodnota vyšší, k součtu přičtu $+1$. Zkoumaná proměnná má nejvyšší hodnotu 10. Místo

```
soucet=0
for k in [2,5,10,8,7,8,0]:
  if k>=5:
    soucet=soucet+1
  else:
    soucet=soucet-1
print(soucet)
```

můžeme použít pole, ze kterého si hodnoty přebíráme, tedy

```
pole=[-1]*5+[1]*6
soucet=0
for k in [2,5,10,8,7,8,0]:
  soucet=soucet+pole[k]
print(soucet)
```

Pracujeme s různě velkými čísly, ne vždy je tento přístup možný, ale je to jedna z optimalizačních metod.

## 8. Rekurze

Řada mých kombinatoricky orientovaných úloh vyžaduje rekurzi. Není to častokrát možné s ohledem na počet možností (omezení pevných disků), ale z rekurze se dá udělat výpis. Jednotlivé možnosti se pak mohou dál analyzovat, to už jde mimo rekurzi. Tohle do určité míry dává naději na užití vGPU do budoucna.

U mne bylo reálnější si u rekurze nechat vypsat řešení do určité hloubky, neb počty možností jsou obrovské hodnoty. To pak předávat na další počítání. Tímto jsem se rekurzi sice nevyhnul, ale podařilo se zadanou úlohu distribuovat. Toho jsem využil u výpočtu počtu obarvení v dřívějších částech.

Školní příklad by byl, kdybychom si u šachu nechali vypsat všechny možnosti pro prvních pět tahů. A to pak dále analyzovali, každou možnost samostatně.

Do mé sbírky možných úloh na řešení přistál tento domácí úkol do profesora Vaška Chvátala, `https://users.encs.concordia.ca/~chvatal/queengraphs.html`. Tedy najít obarvení 27 barvami u úlohy 27-dam.

## 9. Distribuované výpočty

U reálných projektů nás tlačí termíny. Ať se snažíme optimalizovat kódy, jak to jen jde, v určitém místě se musíme zastavit a předat program na skutečné počítání.

Nemusíme zrovna počítat $\pi$ na 300 triliónů cifer, `https://www.youtube.com/watch?v=BD-AJwqzWsU`, abychom si užili distribuované výpočty.

Je dobré vědět, jestli pracujeme s jádry či vlákny, používám tento příkaz, abych se ujistil.

```
cat /sys/devices/system/cpu/cpu*/topology/thread_siblings_list
  | sort | uniq
```

Jak jsou daná jádra vytížená, na to používám tento příkaz

```
nmon -F soubor.nmon -s4 -c1
```

Poté si `soubor.nmon` analyzuji, a dle toho spouštím výpočty vztažené na konkrétní jádro, užívám:

```
nohup taskset -c číslo-volného-CPU ./můj-program parametr
  1>parametr-běžný.log 2>parametr-error.log && [další kroky] &
```

Distribuci, stav úloh a ukládání výsledků si řídím přes MongoDB, nástupce stále používaného MySQL. Osvědčilo se mi to, byť používám úplně základní příkazy. Pokud jsou výstupy z programů rozsáhlejší, osvědčilo se mi nasypat tyto výstupy do souboru a do databáze si jen uložit název takového souboru. Kdo pracuje s databázemi pravidelně, tak ví, že obrázky není ideální do databáze vkládat,

ale uložit si bokem, a v databázi mít jen název souboru, případně nějaká ta metadata.

## 10. Přehled cen

Má spása u úlohy maxima bloků u neizomorfních sudoku byla nabídka virtuálních CPU na `https://compute.cudo.org`. Nabízí i virtuální GPU, byť to mi zatím zůstává na mé úlohy zahaleno tajemstvím. Je to dané tím, že řadu úloh mám rekurzivního typu, to by se na GPU nemělo hodit, ale jistý si nejsem.

V každém případě představuji ceník za měsíc, na udělání si přehledu, zda-li využít svých počítačů, notebooků či serverů versus zakoupené virtuální servery. U nás je klíčová a rozhodující cena za elektřinu.

V tabulkách berte prosím ceny orientačně, u nabízených datových center se liší. Zvolil jsem gb-manchester-1, který mi běžel výtečně. Kvůli kontrole řádků přes `nmon` jsem počet nad 48 CPU nevolil. Také když virtuální stroj spadne, a než se zpátky nahodí, leze to do peněz. Některá datová centra nabízí slevu při zarezervování CPU nad určité období. To je však potřeba znát charakter počítané úlohy.

| vCPU | USD | Za kus | vCPU | USD | Za kus | vCPU | USD | Za kus |
|---|---|---|---|---|---|---|---|---|
| 1 | 8 | 8,00 | 12 | 48 | 4,00 | 96 | 352 | 3,67 |
| 2 | 12 | 6,00 | 16 | 62 | 3,88 | 128 | 467 | 3,65 |
| 3 | 15 | 5,00 | 24 | 91 | 3,80 | 192 | 699 | 3,64 |
| 4 | 19 | 4,75 | 32 | 120 | 3,75 | 256 | 930 | 3,63 |
| 6 | 26 | 4,33 | 48 | 178 | 3,71 | 384 | 1393 | 3,63 |
| 8 | 33 | 4,14 | 64 | 236 | 3,69 | | | |

U datového centra jsem si vždy naklikal, kolik mají aktuálně volných vCPU, snažil jsem se nejít přes polovinu, neb mé výpočty byly intenzívní, aby to jejich servery snášely dobře (minimalizace Stealing Time).

Pro úplnost i přehled vGPU.

| vGPU | USD |
|---|---|
| RTX A500 | 265 |
| V100, A40 (graphics i compute mode) | 293 |
| RTX A6000 | 337 |
| A800 PCIe | 594 |
| L40A (graphics mode) | 646 |
| L40A (compute mode) | 851 |
| A100 80GB PCIe | 1107 |
| H100 SXM | 1655 |
| H100 NVL | 1815 |

Při mazání virtuálního stroje po dokončení výpočtů je potřeba si vždy položit otázku, co vše je potřeba si zazálohovat. Plus si zkontrolovat, že jsou výsledky v databázi. To na domácích strojích člověk neřešívá.

## 11. Na co si dát pozor?

U časově náročných úloh a nutnosti distribuce na víc strojů je potřeba si hlídat řadu věcí.

Pokud jsou virtuální výpočetní stroje vypůjčené či zakoupené, je potřeba dát si pozor na bezpečnost. Konkrétně na vzdálených strojích dáváme přístup do databáze, kopírují se soubory atd.

Je dobré si poznačit instalační kroky, aby se z čisté linuxového distribuce naše výpočty rozjely. A vždy si zkontrolovat, že výpočty už běží. Já obvykle začínám takto (Ubuntu):

```
$ sudo apt update && sudo apt upgrade -y
  # [restart virtuálního stroje]
$ sudo apt install python3 python3-dev python3-pip
  nmon net-tools libmariadb-dev
$ pip3 install mariadb --break-system-packages
```

S tím, že obecně `--break-system-packages` není doporučené, užívají se v Pythonu virtuální prostředí.

Pozn. Příjemné bylo zjištění, že když člověk instaluje program ze zdrojových kódů a zapíše `./configure --` a dá Tab, tak se vypíše seznam možností.

Padá vše. V datových centrech padá elektřina, vyžadují údržbu, ať už plánované či mimořádné zásahy. Na domácích sítích padá internet. Je dobré si ohlídat menší výpadky, ale i ty delší. Menší práce je potřeba znovu nahodit, obvykle zasáhnout do databáze, že je úlohu potřeba spočítat znovu. U větší práce je dobré si zvolit nějakou metodu průběžného ukládání, aby se z toho místa dal výpočet obnovit.

U řady virtuálních serverů jsem se setkal s tím, že mi administrátoři/majitelé rušili úlohy. To se mi u `https://compute.cudo.org` nestávalo, ale je potřeba být na to duševně připravený.

Hodně zdaru u Vašeho počítání!

## Kontaktní adresa

**Ing. Pavel Stříž, Ph.D.,** Nakladatelství Martin Stříž, U Škol 940, Bučovice, okres Vyškov, 685 01, Česká Republika,
*E-mailová adresa*: `pavel@striz.cz`